Seance 4:

Interface de contrôle de haut niveau: l'API.

(durée max: 2h)

Nous allons désormais implémenter les méthodes de haut niveau du projet.

➔ **Observez** le fichier `api_echec.c` et `api_echec.h`.

Implémentation de l'API générale:

La désignation d'**API** indique que ce sont les fonctions de ce fichier qui vont servir à interfacer le moteur du déplacement de pièce avec tout autre programme ou librairie.
En d'autre termes, vue de l'extérieur, un programmeur n'a besoin d'accéder qu'aux fonctions décrites dans l'**API** pour réaliser un jeu d'échec jouable.

Les fonction d'écriture et de lecture on dejà été implémentées.
La fonction `api_echec_deplacement_piece` est la fonction de gestion du déplacement des pièces et doit être écrite.

➔ **Observez l'algorithme** que doit satisfaire cette fonction.

➔ **Complétez** cette fonction.

Pour cela, vous utiliserez les fonctions déjà implémentées de `echiquier`. Vous utiliserez également les fonctions prévues de `echiquier_deplacement`, ainsi que la structure stockant les paramètres d'une erreur éventuelle.
Celles-ci sont encore à implémenter, cependant elle permettent d'établir la **structure de haut niveau** du fonctionnement du programme.

Notez que vous gérerez les **deux joueurs** dans cette fonction, les jeux à étudier dépendent donc du **joueur courant**.

Notez que dans cette fonction, vous ne devez pas manipuler de structures de plus bas niveau tel que le *type de pièce*, ou le *tableau* stockant les pièces.
Votre code doit ici être écrit de manière **indépendante du type de stockage** choisi. Cela permet de potentiellement modifier ce stockage de bas niveau dans le futur sans avoir à modifier votre code de gestion du jeu.

Si un déplacement est **valide**, l'affichage en ligne de commande sera le suivant:
`J.n deplace (x0,y0) => (x1,y1)`

avec:
n=le numéro du joueur (0 ou 1)
(x0,y0)=coordonnées de départ
(x1,y1)=coordonnées d'arrivées

Si les **coordonnées de départ** sont **invalides**, on affichera:
`Coordonnees de depart (%d,%d) invalides`

Si les **coordonnées d'arrivées** sont **invalides**, on affichera:
`Coordonnees d'arrivees (%d,%d) invalides`

Enfin, si il n'existe pas de pièces du joueur courant aux coordonnées de départ choisies, on affichera:
`Pas de piece courante aux coordonnees de depart`

Gestion du déplacement:

➔ **Observez** les deux fonctions des fichiers `echiquier_deplacement`.

`echiquier_deplacement_est_valide`, est la fonction permettant de définir s'il est possible de réaliser le déplacement entre deux coordonnées.

Cette fonction en elle même doit dépendre du **type de pièce**, cependant il est avantageux de cacher cette complexité en proposant une **interface unique** de déplacement. C'est le rôle de cette fonction. En d'autres terme, la fonction sert de *façade* pour rediriger l'appel vers la fonction de vérification qui dépend de la pièce actuelle.
Nous implémenterons cette fonction plus tard, pour l'instant, laissez le code à "return 1".

La fonction de **`echiquier_deplacement_realise`**, permet d'effectuer réellement le déplacement d'une pièce.
Notez que cette fois le pointeur vers l'échiquier courant n'est **pas constant**, car les coordonnées d'au moins une pièce vont devoir être **modifiées**.

➔ **Implémentez** la fonction de déplacement effectif d'une pièce vérifiant les caractéristiques décrites.

➔ **Testez** votre programme. Celui-ci doit désormais etre fonctionnel et pouvoir être interfacé avec le visionneur.

➔ **Testez** certains de vos **fichiers de tests**, et notez que les erreurs de déplacement propres à un type de pièce ne sont pas pris en compte. Par contre, les autres comportements (dépassement des limites du jeu, existence de pièce à l'arrivée, …) doivent quant à eux être correctement pris en compte.

➔ **Implémentez** désormais la fonction **`echiquier_deplacement_est_valide.`** Vérifiez les **pré-requis**. Puis commencez à implémentez les fonctions de déplacement propre à chaque pièce.

Commencons par implémenter le **déplacement du cavalier**.
Pour cela, on implémentera la fonction:

```
int echiquier_deplacement_est_valide_cavalier    (const echiquier*
echiquier_courant,const piece* piece_a_deplacer,int
x_destination,int y_destination,code_erreur_deplacement *erreur);
```

Cette fonction, **spécifique** d'une pièce donnée, ne doit **pas être visible** depuis l'API de haut niveau. Pour cela, nous allons définir ces fonction dans des **fichiers distincts**.

➔ **Créez** les fichiers:
`echiquier_deplacement_privee.c`, et
`echiquier_deplacement_privee.h`.

Ces deux fichiers vont contenir les **implémentations privées** du déplacement propre à chaque pièce.

En séparant ces implémentations de l'appel générique standard de "`echiquier_deplacement`", cela permet de:

- Faciliter **l'évolution** du code si l'on doit modifier le comportement d'une pièce.
- Permettre de compiler et diffuser une **librairie** implémentant le déplacement des pièces sans avoir à diffuser le code source du corps de ces fonctions.
- Permet de mieux **séparer** les appels de haut et de bas niveau de votre projet.

Implémentons la fonction de déplacement du cavalier dans ces fichiers.

➔ Quels **contrats** satisfait cette fonction de déplacement? **Complétez** l'en tête avec ce contrat.

➔ **Implémentez** le corps de la fonction.

➔ **Testez** votre fonction en déplaçant des cavaliers dans le jeu. Vérifiez les cas valides, vérifiez également la bonne récupération du type de l'erreur dans les cas de déplacements invalides.

Fonction de déplacement propre:

(durée max: 2h)

➔ **Implémentez** les autres fonctions de déplacements: *pions, tour, cavalier, fou, roi, reine*.

➔ **Testez** les déplacements au fur et à mesure.

Travail en autonomie.

- Révision (1h)
- Rendu des fonctions de déplacement (2h)

➔ **Achevez** les fonctions de déplacement de pièces.

➔ **Déposez** votre projet.